RL-TR-96-172
Final Technical Report
October 1996

CREST DATA LIBRARY

PAR Government Systems Corporation

Sponsored by
Advanced Research Projects Agency
ARPA Order No. C242, Amend 00

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-172 has been reviewed and is approved for publication.

APPROVED: Stanley E. Borek

STANLEY E. BOREK
Project Engineer

FOR THE COMMANDER: Donald W. Hanson

DONALD W. HANSON, Director
Surveillance & Photonics Directorate

CREST DATA LIBRARY

Contractor: PAR Government Systems Corporation
Contract Number: F30602-95-C-0062
Effective Date of Contract: 15 March 1995
Contract Expiration Date: 15 March 1996
Short Title of Work: CREST DATA LIBRARY

Period of Work Covered: Mar 95 - Mar 96

Principal Investigator: John Reilly
Phone: (315) 738-0600 x-527

RL Project Engineer: Stanley E. Borek
Phone: (315) 330-7013

Approved for public release; distribution unlimited.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Stanley E. Borek, Rome Laboratory/OCSA, 26 Electronic Pky Rome, NY 13441-4514.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1996	Final Mar 95 - Mar 96

4. TITLE AND SUBTITLE

CREST DATA LIBRARY

5. FUNDING NUMBERS

C - F30602-95-C-0062
PE - 63226E
PR - MSEF
TA - BA
WU - 3C

6. AUTHOR(S) John Reilly, R. Patrick O'Connor, Audrey Copperwheat (PAR Government Systems Corp); David Quinn-Jacobs (Request Technologies, Inc.)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

PAR Government Systems Corporation
8383 Seneca Turnpike
New Hartford, NY 13413-4991

8. PERFORMING ORGANIZATION REPORT NUMBER

PGSC Report 96-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Rome Laboratory/OCSA
26 Electronic Pky
Rome, NY 13441-4514

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-172

11. SUPPLEMENTARY NOTES

RL Project Engr: Stanley E. Borek
OCSA/315 330-7013

Sub Contractor: Request Technologies, Inc.
RD #2, Box 395
West Windfield, NY 13491

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release, Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Common Research Environment for Space-Time Adaptive Processing Technology (CREST) Data Library (CDL) was developed to access, query, and retrieve Radar Surveillance Technology Experimental Radar (RSTER) data. Coordinated under the Crest effort, the CDL and the Rome Laboratory Space-Time Adaptive Processing/Advanced Development Tool (RLSTAP/ADT) provide the STAP algorithm developer/researcher with a powerful capability to both access and process RSTER data. Object Oriented (OO) technology was used to develop the design of the CDL. Following OO fashion, prototyping was utilized as the method to verify and refine the design. A result of this development was that the World Wide Web (WWW) was incorporated as a component of the final system. Since the WWW is presently used as a tool by researchers, the WWW CDL prototype has been retained to serve as a tool for the novice user. The other components of the system utilized the Object Management Architecture (OMA) to accomplish the goal of deploying a distributed system. This Final Technical Report is divided into two sections. The first, System Description, defines the overall architecture and concept of operation for the CDL. The second, Technical Goals and Accomplishments, discusses future direction of the WWW and OMA with respect to this project for the concepts of distributed computing, rapid prototyping and supporting/future technologies.

14. SUBJECT TERMS Crest Data Library, Common Research Environment for STAP Technology, Hypertext Markup Language, Common Object Request Broker Architecture, Interface Definition Language.

15. NUMBER OF PAGES 44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Glossary

API	Application Programming Interface
BOA	Basic Object Adapter
CDL	CREST Data Library
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
COTR	Commanding Officer's Technical Representative
CPU	Central Processing Unit
CREST	Common Research Environment for STAP Technology
DII	Dynamic Invocation Interface
DS	Data Store
DSI	Dynamic Server Interface
ftp	File Transfer Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
http	HyperText Transfer Protocol
httpd	HyperText Transfer Protocol daemon
I/O	Input/Output
IDL	Interface Definition Language
IFR	Interface Repository
IMR	Implementation Repository
JDK	Java Development Kit
MHPCC	Maui High Performance Computing Center
NCSA	National Center for Supercomputing Applications
OMA	Object Management Architecture
OMG	Object Management Group
OO	Object-oriented
ORB	Object Request Broker
OS	Operating System
OT	Object Table
PDS	Persistent Data Service
RSTER	Radar Surveillance Technology Experimental Radar
STAP	Space-Time Adaptive Processing
Tcl	Tool Command Language
TCP/IP	Transmission Control Protocol/Internet Protocol
Tk	Toolkit
URL	Uniform Resource Locator
WWW	World Wide Web
XDR	External Data Representation

CREST Data Library Final Technical Report

1. INTRODUCTION

The CREST Data Library was developed to access, query, and retrieve RSTER (radar surveillance technology experimental radar) data. Coordinated under the CREST (Common Research Environment for STAP [Space-Time Adaptive Processing] Technology) effort, the CREST Data Library and RLSTAP/ADT provide the CREST researcher with a powerful capability to both access and process RSTER data. This document is delivered by PAR Government Systems Corporation (PGSC) under the CREST Data Library (contract no. F30602-95-C-0062) in satisfaction of CDRL A005. The format of this document has been developed in cooperation between PGSC and the COTR (Contracting Officer's Technical Representative) at Rome Laboratory/OCSA.

The Final Technical Report is divided into two sections. The first section, System Description, defines the overall architecture and the concept of operations for the CREST Data Library. The technical accomplishments and future direction of the project are described in Section 3.

2. SYSTEM DESCRIPTION

Object oriented (OO) technology was used to develop the design of the CREST Data Library. Following typical OO fashion, prototyping was used as a method of verifying and refining the design. In the course of prototype development, it became evident that the prototype that was developed using the World Wide Web (WWW) could be used as a component in the final system. The WWW-based CREST Data Library prototype has been retained to serve as a tool for the novice user. The WWW is now a widely used tool for people doing research. Therefore, the simple interface to the CREST Data Library presented by this prototype is useful to introduce the project concepts.

The other components of the system used the Object Management Architecture (OMA) to accomplish the goal of a distributed system. This configuration lays the foundation for a truly distributed system that facilitates the sharing of data, algorithms, and enables collaborative development as well.

2.1 System Architecture

The software architecture of the CREST Data Library is best described as a system that has two distinct methods of accessing a common set of data. These are the World Wide Web (WWW) method of access and the Object Management Architecture (OMA) method of access. The architectures of both these components are described in the following sections.

2.2 WWW Architecture

Figure 1 presents the architecture of the WWW component of the system. This is the classic method of database access currently employed on the WWW. This component of the system proved useful in characterizing the important query parameters and the typical usage of the system by the end user.

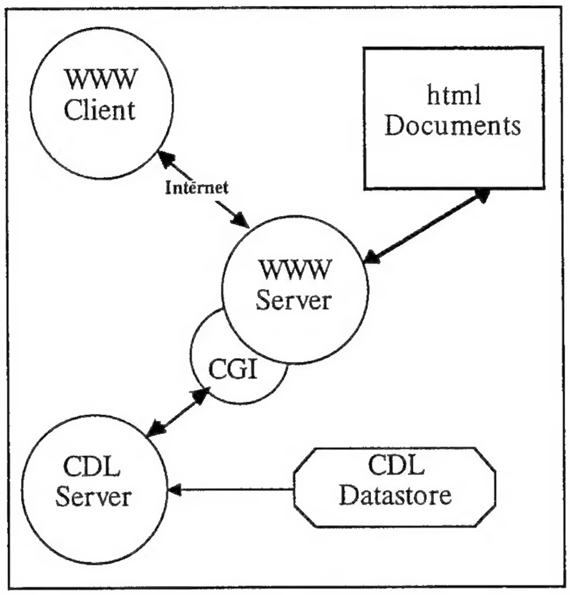

Figure 1. WWW Component Architecture

- WWW Client - A forms-capable WWW browser (e.g., Netscape). All query specification and query results/downloads are made available through this interface.
- WWW Server - An http server (e.g., NCSA httpd). The WWW Server: 1) provides access to html documents that are used to provide ancillary CREST Data Library information, 2) enables the downloading of radar data, and 3) provides access to the CREST Data Library Server (and onto its database) through the use of the Common Gateway Interface (CGI).
- CGI - The CGIs are used to provide access to: 1) the CREST Data Library Server, 2) the application that extracts a CPI from a radar data file, and 3) to construct both the query form and results form.
- CREST Data Library Server - Parses the queries given in the query form, runs the query against the datastore, and returns the data file names that pass the query.

2.3 OMA Architecture

The OMA components of the system were implemented as a precursor to a completely distributed environment. The design of the system calls for the implementation of a federated query mechanism that permits the distribution of the CREST Data Library Server and of the data. Figure 2 illustrates the OMA-based architecture.

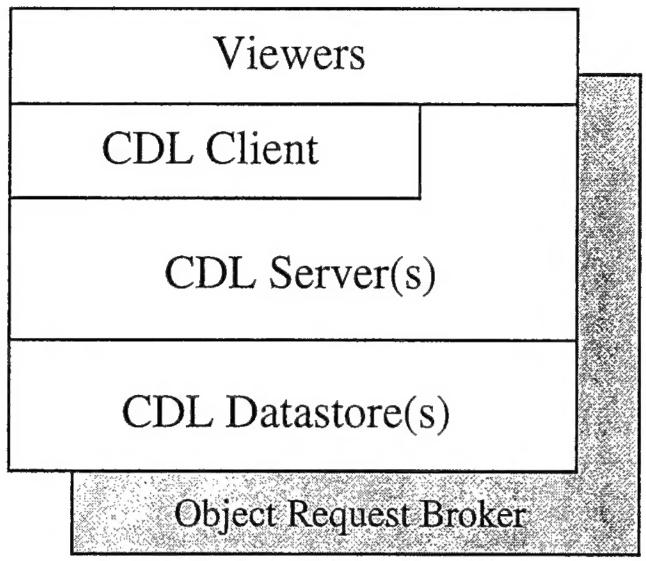

Figure 2. OMA Component Architecture

- Viewers - This element of the system provides the user interface. When the CREST Data Library is initialized, the CDL Client creates a viewer to provide access to the Library. The viewers in turn use the services of other viewers to provide access to query tools, map views, and holdings details.
- CDL Client - Provides the client-side configuration management. It initiates contact with the CDL server and provides the viewers with the context. The CDL Client does not act as an intermediary between the Viewer and the CDL Server.
- CDL Server - Provides the server-side configuration management. The CDL Server "points" to an associated CDL Datastore. In the future, this will facilitate the organization of servers as well as the grouping of datastores.
- CDL Datastore - Provides the ability to query and download holdings. It encapsulates the underlying database management system being employed.

2.4 Concept of Operations

The most effective way to develop an object-oriented application that is easily understood by the target audience is to select a paradigm that is familiar. This facilitates the adoption of the system and allows the user to naturally interact with the information of the system rather than having to concentrate on the method of interaction. The CREST Data Library is modeled after a research library. Figure 3 shows the first screen presented to the user, an overhead view of a library. In the figure, the object identified as #1 provides access to the card catalog. Object #2 in the same figure accesses the help facility for the CDL. All other objects in the interface are currently inactive.

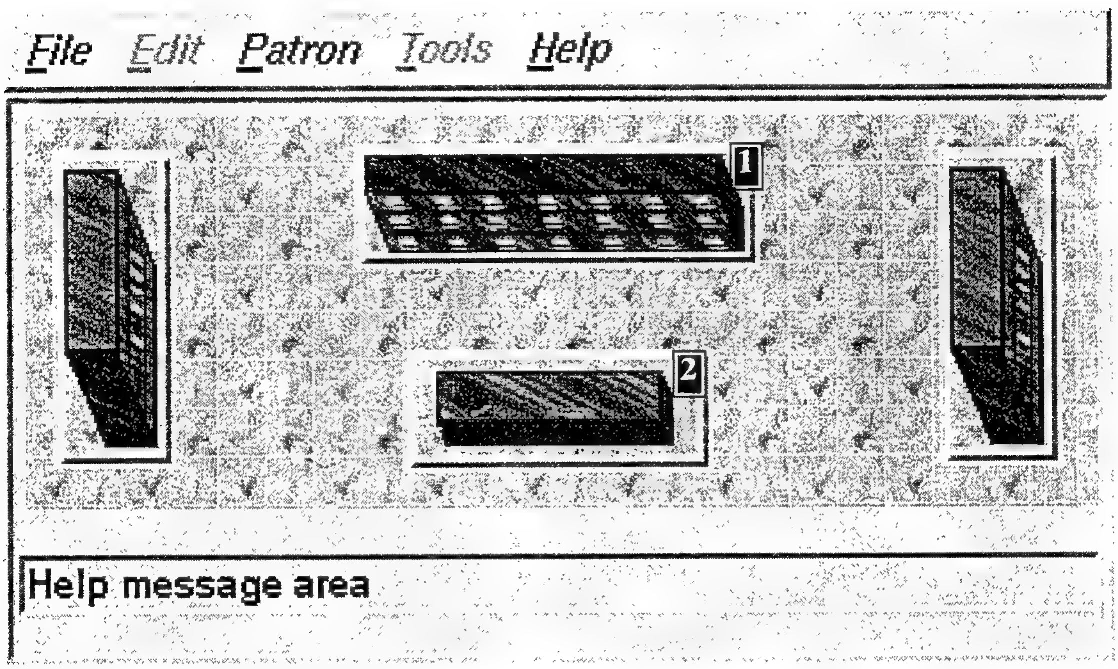

Figure 3. Library Screen

For the purposes of this effort, a research library has the following characteristics:

- The unit of measure of the research library contents is a holding.
- The people that use a library are called patrons.
- A holding may be expressed in any of several media. For example, in a real library, a holding may be a book, a magazine, a video tape, etc.
- Holdings may contain references to other holdings.
- A library may be logically "connected" to other libraries via mutual agreement. Connected libraries may choose to share selected holdings.
- The contents (i.e., all holdings) of a library are recorded in the card catalog. If a library is connected with another library, the shared holdings of the connected library are included in the card catalog.

- Typically, the information contained in the cards of the card catalog are in a fixed format.
- The card catalog can be searched. Typically this is performed electronically, based on the contents of the card describing each holding.
- Holdings are not checked out of a research library. All or parts of a holding may be copied by a patron for other uses.
- Patrons must be registered (have a library card) to use a library.
- Holdings are added to a library only after review by the librarian.

Beyond the real-world definition given above, the current version of the CREST Data Library imposes constraints upon the research library paradigm. For each of the Library's two components (a WWW-based component and an OMA-based component), these constraints are different. The WWW-based component has the following constraints:

- Libraries cannot be connected.
- Patrons only need to register when they extract data from the holdings.
- Card catalog searching is limited to form-oriented querying of Radar Data holdings.
- Adding, deleting and modifying holdings are not supported with a GUI tool and cannot be performed by a Patron. Only those with the proper accounts on the host system can add/delete/modify holdings.

The OMA-based CREST Data Library has the following constraints:

- Libraries cannot be connected.
- The contents of other holdings types can be viewed by a text viewer, which places the tag/value pairs of the holding into a window. This viewer does not provide the capability to query. The user can only use the General Holdings Viewer to query these other holdings types; the tags unique to the other holdings types cannot be queried.
- Adding, deleting and modifying holdings are not supported with a GUI tool and cannot be performed by a Patron. Only those with the proper accounts on the host system can add/delete/modify holdings.

3. TECHNICAL GOALS AND ACCOMPLISHMENTS

Object oriented (OO) technology was used to develop the design of the CREST Data Library. Following typical OO fashion, prototyping was used as a method of verifying and refining the design. In the course of prototype development, it became evident that the WWW prototype could be used as a component in the final system. Thus, the WWW-based CREST Data Library prototype has been retained to serve as a tool for the novice user. Since the WWW is now a widely used tool for people doing research, the simple interface to the CREST Data Library presented by this prototype is useful to introduce the project concepts.

The other components of the system used the OMA to achieve the goal of a distributed system. This configuration lays the foundation for a truly distributed system that facilitates the sharing of data, algorithms, and results and enables collaborative development as well.

The remainder of this section describes three factors that are critical to the success of the CREST Data Library: distributed computing, rapid prototyping, and an understanding of possible future directions.

3.1 Distributed Computing

3.1.1 WWW Distribution

The WWW was conceived and developed by a small group of physicists and engineers at CERN, the European Particle Physics Laboratory in Geneva, Switzerland. The original idea of the "Web" was based on using a hypertext system to allow researchers located at remote sites to collaborate and quickly and easily share information. The WWW has evolved into a global, seamless environment in which all information (text, graphics, images, audio, video clips, and computational services) accessible on the Internet can be accessed in a simple and consistent manner through the use of a standard set of naming and access conventions. Essentially, WWW facilitates use of the Internet in the following ways:

- provides a graphical user interface (e.g., Web browser) for many different platforms
- uses the same communications and underlying support tools as Internet, but hides the complexity of details from the user
- based on common standards and protocols which promotes information sharing
- supports multimedia (images, graphics, text, audio, and video)
- permits easy delivery of information in any format to wide variety of platforms
- permits the user to easily access sites all over the world, by simply clicking on a selection (i.e., highlighted or underlined string of words).

3.1.1.1 Overview Of Web Architecture

The WWW is a collective network of servers residing on the Internet and using a common set of protocols and access mechanisms to communicate among themselves. The main components of the Web architecture are shown in Figure 4. Web clients and servers use the HyperText Transfer Protocol, *http* – a fast and extensible transport protocol for communication within the Web. The *http* protocol is required in order to permit the transfer of hypermedia documents. Clients send messages to Web servers, which are called http daemons, or *httpd*. Web clients also inform Web servers what formats they can handle.

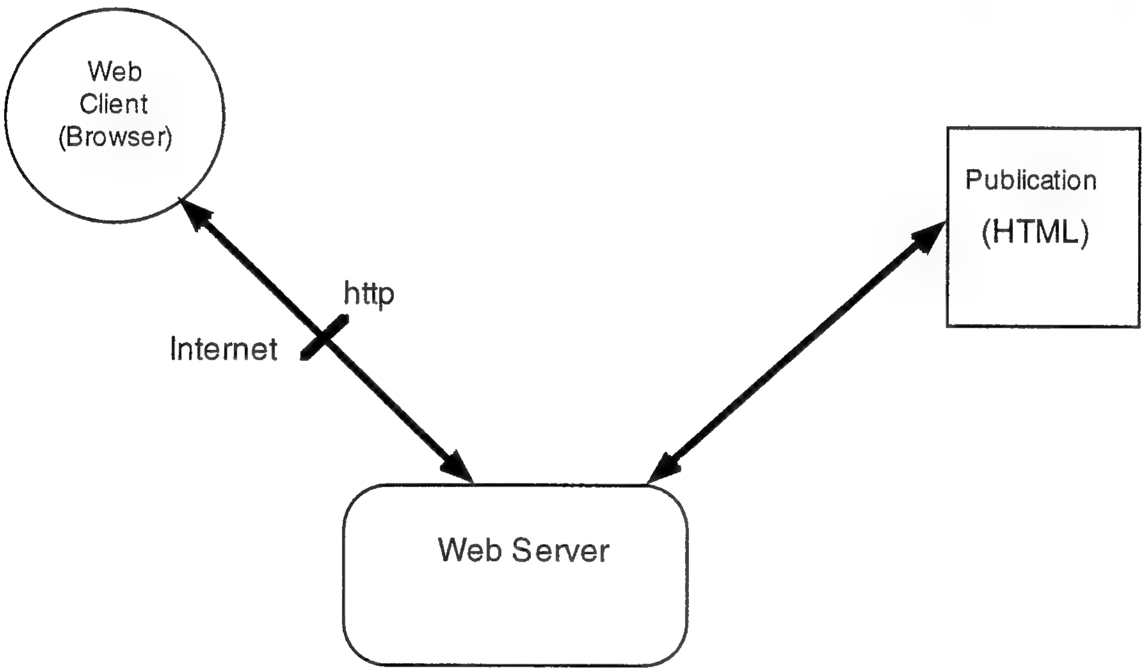

Figure 4. World Wide Web Architecture Components

The Web servers receive messages and are responsible for sending the requested information to the requester (client/browser). Only information that resides on Web servers belongs to the Web. Web servers are responsible for document storage and retrieval. Often the document is comprised of several Web objects (graphics, audio clips, etc.) including the basic text, and the Web server sends each of these objects back to the client individually. The Web browser collects the separate pieces that were requested, interprets the code in the document, and presents the document to the user. Note that a Web page may have references to objects all over the Internet. Web servers may have to pass client requests to search engines and applications located at other servers, in order to satisfy requests. Accordingly, Web servers use the Common Gateway Interface, *CGI*, a standard interface between external gateway programs and *httpd* and other information servers.

The document is represented in a common standard language called HyperText Markup Language, or *html*. All Web browsers are able to interpret *html*. The *html* is a simple language for creating and translating Web documents. The *html* documents consist of formatting codes that specify document layout (fonts, titles, paragraphs, bullet lists, etc.) and hyperlinks.

Network-wide addressing is accomplished by a standard addressing scheme called the Uniform Resource Locator, or *URL*. Every retrievable piece of information on the Web is identified by a unique *URL*, which can be viewed as the networked extension of the standard filename concept. The *URL* can represent any file or service on the Internet and its syntax includes the name of the object, where it is located, and the protocol used to access it.

3.1.1.2 CREST Data Library On The World Wide Web

During the initial phase of the effort, the capability to access CREST data using standard Web tools was developed. Standard *httpd* Web server software was installed at the Maui High Performance Computing Center (MHPCC), in order to establish the CREST Web site. Using the *html* language a CREST home page and standard query form was developed. Hence, Web clients using standard browsers (e.g., Netscape, Mosaic) may connect to the initial prototype via *URL*: `http://wwwcrest.mhpcc.edu/`.

Shown in Figure 1 is the Phase 1 CREST Data Library (CDL) prototype architecture. The CDL server was derived from the Rome Laboratory/IRRE IE2000 Imagery Server legacy software. The metadata for the CDL server was populated for the RSTER radar database. The Web server and CDL server communicate through the CGI interface. The html standard query form associated with the CREST home page was based strictly on parameters for the RSTER database.

3.1.2 Object Management Architecture

The object-oriented approach to distributed computing is a natural step forward from client-server systems of today. In support of this claim, most standardization efforts in heterogeneous distributed systems are based on the object model. In fact, standardization of object-oriented distributed computing appears to be converging on the Common Object Request Broker Architecture (CORBA) of the Object Management Group, Inc. (OMG), which is an international standardization consortium supported by over five-hundred (500) information systems vendors, software developers, and users. OMG's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. OMG's primary goals are to promote standards for reusability, portability, and interoperability of object-based software in distributed heterogeneous computing environments.

The discussion which follows provides an overview of a CORBA-based distributed object architecture, as well as a detailed discussion of a CORBA-compliant object-based software product (IONA Technologies' Orbix) used during development of the CREST Data Library.

3.1.2.1 Object Management Architecture Overview

A fundamental concept in the distributed object environment is that of "object." An object is a grouping of computational operations and data into a modular unit. An object is defined by its external interface, its behaviors when that interface is invoked, and its state. An interface is a description of the set of possible operations that a client may request of an object. An object that complies with a specific interface is said to be of that interface type.

The OMG is establishing standards for integrating distributed applications through the use of object technology. The proposed architecture, the Object Management Architecture (OMA), is depicted in Figure 5. In the OMA, every piece of software is represented as an object. Objects communicate with other objects through the Object Request Broker (ORB). Figure 6 illustrates the process whereby a client, wishing to perform an operation on an object, sends a request through the ORB to Object Implementation. Object Implementation represents the code and data that actually implement the object. The basic components of the ORB architecture based on the OMG's OMA are explained in the following paragraphs.

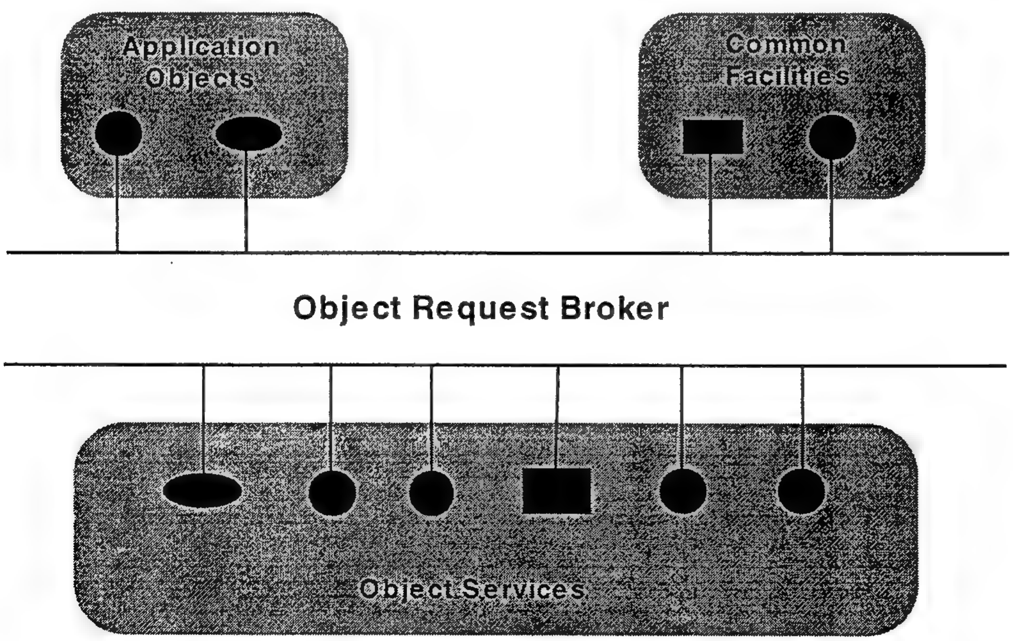

Figure 5. Object Management Architecture

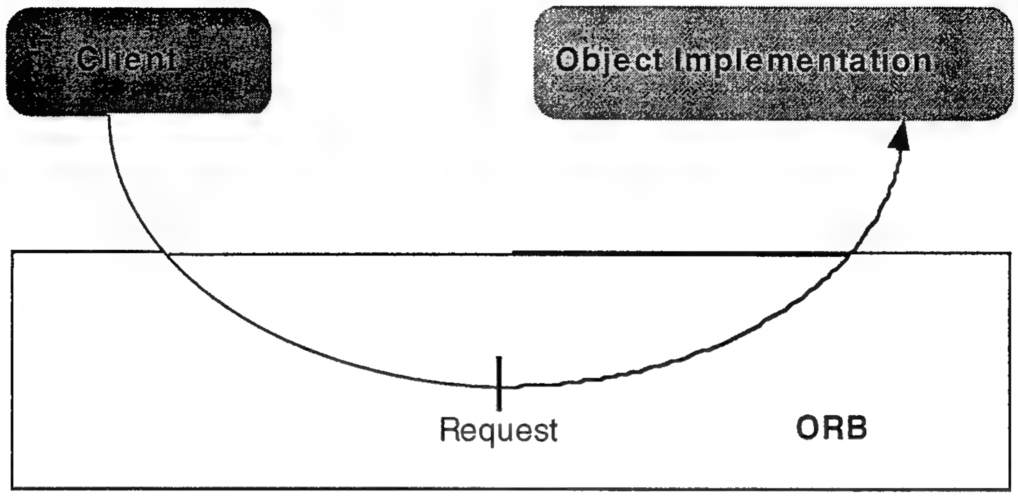

Figure 6. Sending Request Through ORB

3.1.2.1.1 Object Request Broker

The ORB is a group of processes in the distributed environment that enable ORB objects to communicate. The ORB connects objects requesting services to the objects providing them, and it activates object servers when client requests need to be serviced. Accordingly, the ORB enables objects to transparently make and receive requests and responses in a distributed environment. The ORB mediates between clients and implementations (application objects) by providing a specified interface to such clients, and another interface to such implementations.

The ORB is the foundation for building applications from distributed objects and for interoperability between applications in heterogeneous environments. The proposed standard for the ORB, Common Object Request Broker Architecture (CORBA), supports a general Interface Definition Language (IDL) that may be mapped to a programming implementation language (e.g., Ada, C, C++). Applications may be implemented with "wrappers" to provide CORBA-compliant behavior. A wrapper is an interface mechanism that defines an object interface to the object-oriented components of a system for an element (application, API, etc.) that does not have an object interface.

3.1.2.1.2 Object Services

The ORB is supported by a collection of services that includes both interfaces and objects. They support the ORB by providing the functions necessary for object implementation. These services are indispensable for building any distributed application because they furnish the basic set of interfaces, protocols, and policies used in building distributed applications. Object services must be independent of any application domain.

3.1.2.1.3 Common Facilities

Common facilities are made available through OMA-compliant interfaces. These are common general services that many applications may share, and that are not as fundamental as Object Services.

3.1.2.1.4 Application Objects

Application objects are developed utilizing these ORB architecture components. Application objects are the products that are built "on top of" the ORB architecture components; the interfaces for these objects are application-dependent and are controlled by the developers of the application. Application objects belong to the specific applications that are being integrated in the distributed environment. Application objects can communicate throughout the network using the ORB. In order to develop meaningful and effective communications between application objects, application interfaces and protocols must be compliant with other participating application objects within the distributed environment (and vice versa). Once an object is created, its services are available to other objects on the network.

3.1.2.2 Orbix Architecture

IONA Technologies is one of the leading developers of distributed object request broker technology based on the OMG CORBA standard. IONA designed and developed the Orbix software (written in C++) to be fully CORBA-compliant with emphasis on flexibility and performance. Orbix supports C and C++ language bindings and C++ mapping for IDL.

Orbix is primarily implemented: 1) as a pair of libraries — one for clients and one for servers; and 2) as an activation daemon (i.e., *Orbixd*). The client library is a subset of the server library, which can both send and receive remote object operation requests; the client library is restricted to only issuing such requests. *Orbixd* performs (re-)launching server processes dynamically as required; it need only be resident at server nodes. *Orbixd* uses a simple database, called the Implementation Repository (IMR), to access activation information for all of its object implementations. For each object implementation, the IMR contains a CORBA activation mode, the name of the associated executable image file, and any command line parameters.

The Orbix ORB is not a distinct centralized computational entity through which all object requests must pass. Because of the Orbix library implementation, the ORB is conceptually everywhere; object requests are directly from client code to the invoked object implementation.

In addition to the libraries and daemon described above, Orbix also provides an IDL compiler, Interface Repository (IFR), and several utilities for managing the IMR. The IDL

compiler is used during development to translate CORBA IDL definitions into IDL stub code to facilitate remote operations. The IFR is an Orbix application that permits other applications to determine the interface properties of object implementations during runtime. The Orbix Architecture is illustrated in Figure 7. Figure 8 conceptualizes how the components of the ORB architecture fit together from the standpoint of logical interfaces. The Orbix architecture is discussed, with reference to the structure diagrams in Figures 7 and 8, in the following paragraphs.

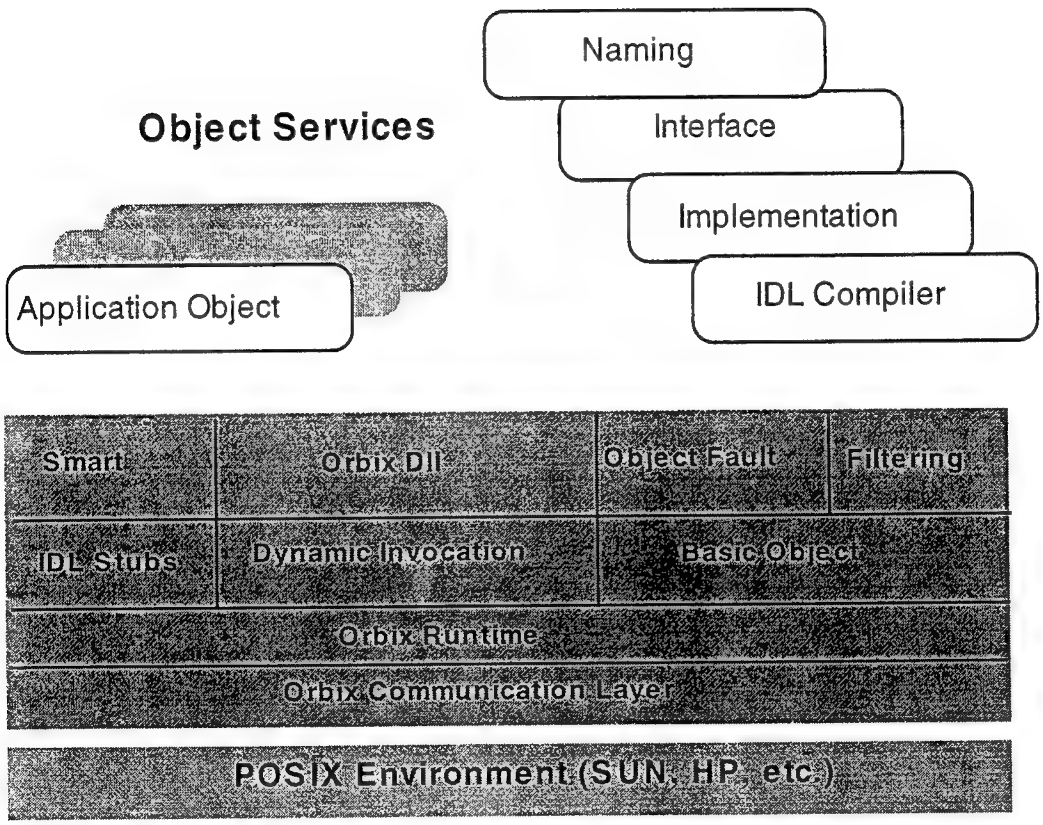

Figure 7. ORBIX Architecture

3.1.2.2.1 Orbix Architecture – Communication Layer

In the Orbix Communication Layer shown in Figure 7, the basic transport facilities are performed by four classes: *MediaAccess*, *Network*, *RequestSender*, *RequestReceiver*. The default implementation of the basic classes, *Network* and *MediaAccess*, uses TCP/IP (Transmission Control Protocol / Internet Protocol) and XDR (external data representation) encoding along with a simple message protocol. *RequestSender* and *RequestReceiver* classes are used to implement messaging of object requests, and both classes use the basic transport classes. Both of these classes rely on the UNIX select system call. The file descriptors used in the select may be merged with descriptors used in other subsystems

(i.e., X Windows). The four classes of the communication layer interface with the Orbix runtime layer, and are represented as the ORB-dependent Interface abstraction in Figure 8.

3.1.2.2.2 Orbix Architecture – Runtime Layer

The runtime layer implements the following CORBA-compliant interfaces as shown in Figure 8:

- Client application programming interfaces (APIs), represented by the dynamic invocation interface (DII).
- IDL messaging stub code that is produced by the IDL compiler from IDL source descriptions.
- Server APIs (i.e., basic object adapter (BOA), dynamic server interface (DSI), and IDL skeleton).

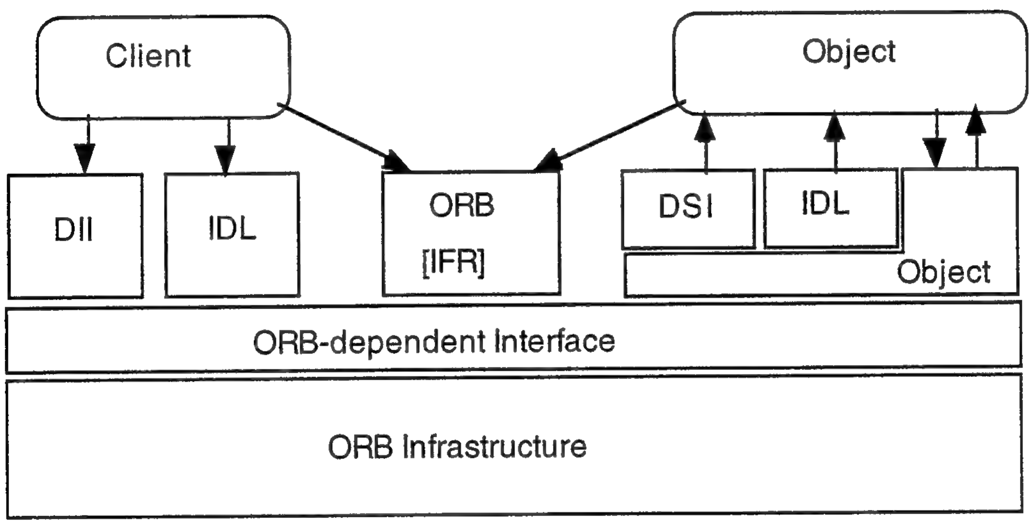

Figure 8. ORBIX Interfaces

On the client side, either the DII or IDL stubs are used to request server invocations. On the server side, either IDL skeletons or DSI are used to receive invocations on objects through the ORB infrastructure and the BOA.

The choice of which type of interface to use for the client or server depends on whether static or dynamic typing is used. Dynamic typing means that types (i.e., IDL interfaces) can be dynamically added and changed at runtime rather than statically defined at compile time. When static typing is used, IDL source descriptions are processed by the IDL compiler to generate IDL stubs and skeletons with which to program. If dynamic typing is used, the Interface Repository (IFR) provides the information that the IDL compiler would obtain directly from IDL definitions. Then, on the client side, the DII makes calls as a stub would, and on the server side, the DSI handles calls in the same manner as a skeleton. In

order for the dynamic interfaces to work, it is imperative that all objects implemented with static typing be registered with the IFR.

The Interface Repository (IFR) is part of the common ORB interface (as shown in the middle of Figure 8) that can be used by both clients and servers. The IFR contains IDL definitions that are used for dynamic invocation calls and that require navigating through the available interfaces.

The basic object adapter (BOA) is responsible for generating object references. The BOA permits the ORB to locate, activate, and invoke operations on an object; and it provides the interface for object implementations (servers) to access ORB functions. Note that, when using the BOA, it is not necessary to use the IFR.

In Orbix, the fundamental runtime classes are the *Request* and *Object* classes. The *Request* class implements the CORBA *Request* interface and is part of the DII. It is also used by generated IDL stub code. The *Request* class checks the invocation arguments, and takes appropriate marshaling action depending on whether dynamic or static typing is invoked. In general, if CORBA static invocation interface is used, the generated IDL stub code insulates programmers from the differences between *Request* and *Object* classes.

The *Object* class implements the CORBA Object interface. Accordingly, an instance of the *Object* class has the fundamental information to communicate with a remote object. When an IDL interface is compiled with the IDL compiler, a corresponding C++ class, called an IDL class, is generated. The *Object* class is the base (highest-level) class of all IDL classes.

Orbix, during runtime execution, also builds a "proxy" (or surrogate) for each remote object used by the local process. A proxy is an instance of an IDL class, and the runtime maintains a table of all proxies and of all implementations of IDL interfaces; this table is called the Object Table (OT), and a single OT is built for each UNIX process. The proxy is an implementation that is unique to Orbix.

3.1.2.2.3 Orbix Architecture – Smart Proxies

The default action of each IDL stub is to marshal client requests and to forward them on to the remote object server. The stubs are the methods of the proxy objects, which are instances of IDL classes. However, the default action can be changed by overriding the generated IDL stub code via a new derived class. In general, for a specific IDL class, there can be several alternative derived classes. When a new proxy must be constructed at runtime, these classes can collaborate to determine which of them is most appropriate for this particular proxy construction, based on the identity of the specific remote object for which the proxy is to be built.

Smart proxies allow the behavior of remote representatives of objects to be extended and modified. Smart proxy support for a particular IDL interface is often provided by the server programmer, in order to control the behavior of the server as it is presented to its clients in their process contexts. Typically, smart proxy support is transparent to the client programmer. Examples of where smart proxy support may be used include:

- Notification of a change in server status via server "call-back" to a smart proxy.
- Server rebinding, where the proxy can be rebound to an alternative remote server when the original server fails.
- Breakpoints for debugging and executing trace code.
- Type conversion of IDL types to non-IDL types, when migrating legacy applications onto the CORBA standard (i.e., wrapper development support).

3.1.2.2.4 Orbix Architecture – Filtering

The Orbix runtime permits programmers to supply filtering code in both clients and servers. Filters are instances of filter classes, which are derived classes of the Orbix class *Filter*. Filters are formed in a linked list, and an arbitrary number of filters may be installed. Basically, filters are applied when an operation request or reply is about to be transmitted from a UNIX process context, and when such a request or reply is about to be received. The key parameter associated with each filter event is the current request/reply, from which the target object and operation name can be determined. Further parameters can be marshaled into the current request/reply by the filter. Having processed an event, a filter can choose to pass the event to the next filter in the chain or to suppress the event from the remaining filters. Filtering may also be performed on a specific object instance, with the actual parameters of the specific object operation being unmarshaled and directly available to the filter code. The filter code, therefore, is another implementation of the IDL interface associated with the target object; the filter can have methods for each of the IDL operations of the target object.

Filtering has similar uses to those of smart proxies, particularly providing wrapper support for legacy applications. Combined with the capability to mediate requests by smart proxies, filtering provides a powerful mechanism for tailoring a distributed object environment for both clients and servers to dynamically changing circumstances.

3.1.2.2.5 Orbix Architecture – Object Fault Handler

Orbix does not provide support for handling persistent objects, which are those objects whose state can be saved and restored from auxiliary (disk) storage. However, coupling a persistent store regardless of its associated data model is an important requirement for many applications. Orbix runtime does provide fundamental support for this by means of the

abstract class, *LoaderClass*. The instances of *LoaderClass*, which are loaders, are formed in a linked list.

Whenever a new object (described in IDL) is compiled and registered with Orbix, the loaders are notified. The server programmer can name a new object and pass this name to the loaders along with the identity of the object's IDL interface. Typically, there is one loader for a particular set of IDL interfaces. The loaders must be coded so that they agree on which loader is responsible for which object. The loader responsible for a new object can choose to adopt the proposed object name (if any) or generate a new name for the object. Note that a generated name might be a database key, which will later be used for storing and retrieving the object.

When a server process exits, the loaders are notified and can safely store the state of the objects for which they are responsible. Similarly, when an operation request is received into a server, the loaders are notified if the Orbix runtime cannot locate the target object (i.e., target object is not yet registered in the OT). This is considered an "object fault." The target object's name is passed to the loaders, in order for the responsible loader to be identified and subsequently attempt to restore the object's state from persistent storage. If the target object is successfully retrieved and restored, the OT is updated and the operation request is resumed (transparently). Alternatively, if the target object cannot be retrieved, or no loader recognizes the object's name, an exception condition is returned back to the client.

Orbix intentionally does not provide persistent storage management. Rather than burden the ORB with this requirement, the actual storage and retrieval of a specific object into and from its persistent state is considered to be more appropriately handled by software optimized for each particular storage manager. Thus, the integration of storage handlers and the development of application-specific loaders is the responsibility of the Orbix developer.

3.1.2.2.6 Orbix Architecture – Object Naming and Location Service

In Orbix, the name associated with a server is significant because it is registered in the Implementation Repository (IMR), and it is used by the activation daemon, *orbixd* to identify the executable file to activate the server. The default name of a server is the same as that of the IDL interface which it implements. However, it is possible for a server to implement several IDL interfaces. In this case, the server name may be associated with a master IDL interface that abstracts the entire functionality provided by the server. Finally, server names can be independent of any IDL interface; and they may be hierarchically structured similar to UNIX path/file names.

The server name is also important for obtaining all objects that are maintained by the server. In Orbix, an object is named by concatenating the host name of the node at which it

was created with the name of the server which created it, and the name which the object has within that server (i.e., the object's marker name).

Most importantly, when a client program wishes to use a named service at runtime, the client must instruct Orbix to bind the client to an applicable server. The CORBA specification calls for the client to provide Orbix with the full Orbix object reference. Alternatively, in Orbix, a client may bind to a specific server name at a particular host, where the server name has been previously registered in the IMR at that host. The client may also bind to a specific named object at that server, as directed by the object's marker name. If the client does not specify a target marker name, Orbix binds the client to any object within the server which provides an interface compatible with that expected by the server.

The most typical situation of a client bind is when the client identifies a service, but not a specific host that can provide that service. The service is named by a server name, and an arbitrary number of hosts may recognize that server name, based on the contents of their respective IMRs. In this case, Orbix must "search" the network to identify suitable hosts. This search is facilitated by the location mechanism, which is implemented in the Orbix runtime via the abstract class *locatorClass*. This class is called (transparently to the application code) with a service name and returns a list of names of hosts on which the service resides. The default implementation of *locatorClass* uses a configuration file at each host; this file registers information about which hosts, and groups of hosts, can provide specific services. In addition, these configuration files can be linked, and queries can be forwarded to other hosts if the desired information is not found in the current file. Finally, the default implementation can be overridden by providing and registering a derived class of *locatorClass* which might implement a directory for mapping service names to groups of hosts. When multiple hosts are identified by the locator service, Orbix selects one of the hosts randomly.

3.1.2.2.7 Orbix Architecture – IDL Compiler and Interface Repository

The IDL permits programmers to define various interfaces and related data that can be utilized to develop distributed applications. IDL source files are a form of representation for defining interfaces. The IDL compiler performs a front-end parsing and analysis on IDL source interface specifications and generates three back-end interpretations of the parsed IDL code: 1) simple regeneration of the original source IDL; 2) stub code generation for C++; and 3) generation of information for the Interface Repository (IFR). The IDL source language closely resembles C++. To implement an interface, the IDL source is mapped into corresponding elements of C++; IDL operations are mapped into C++ member functions and IDL interfaces are mapped into C++ classes.

As described in Section 3.1.2.2.5, Orbix does not provide a persistent object store. As a result, the IFR uses IDL source files as its persistent storage. In Orbix, the IFR is built

as a normal object server. The IFR uses the object fault handler (see Section 3.1.2.2.5) to detect attempted invocations on objects as yet unknown to it. Then to resolve these object faults, the IFR invokes the IDL compiler at runtime to parse the appropriate IDL source files and generate the objects used internally by the IFR. Since the IDL compiler is dynamically linked into the IFR application and is not called as a separate process, there is virtually no delay in the IDL compiler's activation after an object fault is detected. Normally, however, the IFR (and IDL compiler) are instructed to pre-parse commonly used IDL source interface files so that object faults can be avoided.

A possible concern with the Orbix IFR implementation and the runtime parsing is the demand on the virtual memory requirements of the IFR for long-running systems. In such environments, there is a possibility that the memory usage of the IFR may grow. Orbix is contemplating a mechanism that would monitor IFR memory usage, and deactivating less recently used interfaces.

3.1.2.3 Data Store

One of the most important object services supporting the ORB is the concept of the Data Store as shown in Figure 9. The ORB architecture facilitates the access of clients operating on a network in one facility to data holdings at another remote facility without their having to be concerned with the underlying data storage interfaces at the remote site.

The Data Store is a low-level interface that is hidden from application objects and their clients. Accordingly, the Data Store provides service to any object that indirectly uses the Data Store interface. The Data Store permits interoperation with heterogeneous storage services and provides application objects a means of storing data that are not objects. Thus, the Data Store service provides a layer of abstraction that permits implementations of low-level storage services to be added, improved, and replaced without affecting the clients that use the service. Thus, the Data Store concept facilitates the "plug-and-play" paradigm for heterogeneous databases that is critical for integration in an open architecture environment.

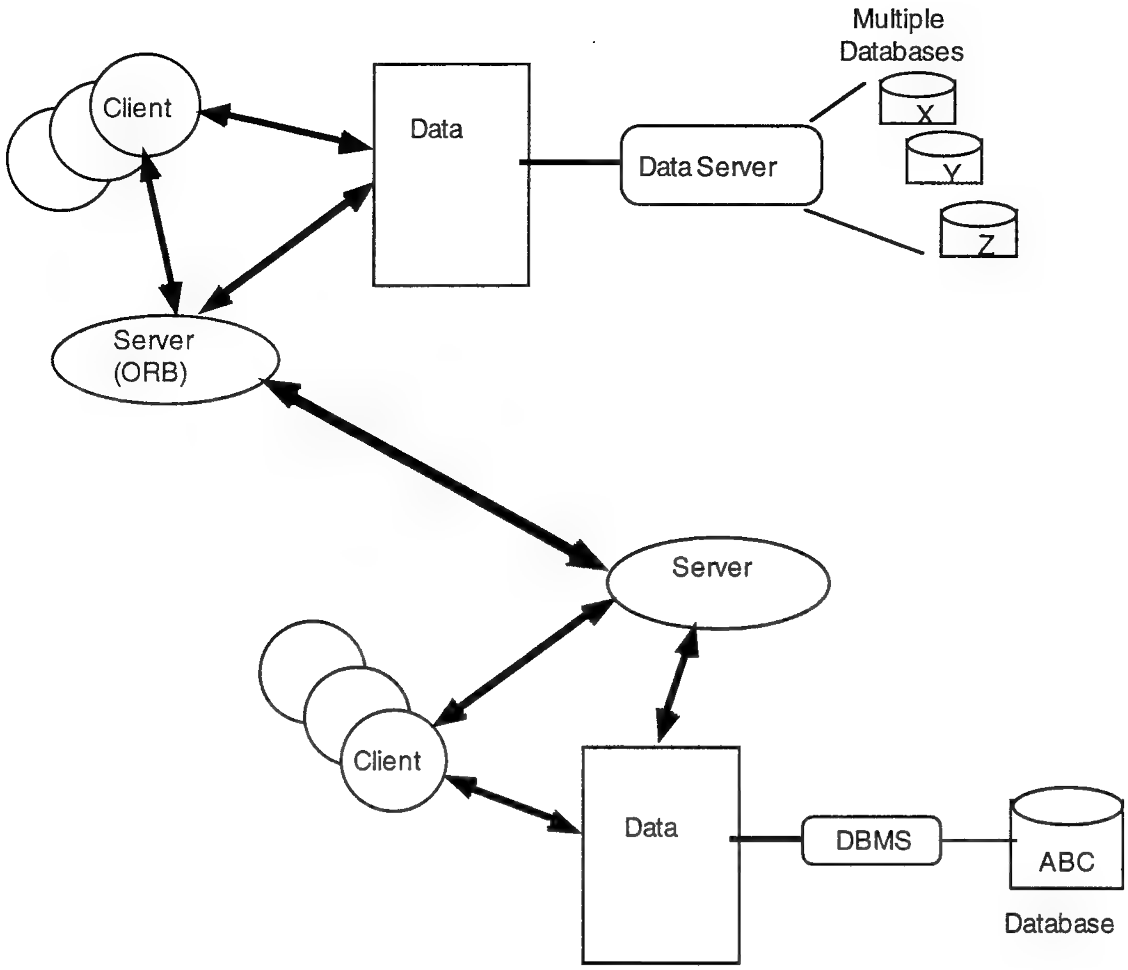

Figure 9. Interoperable Data Stores

Data Store Implementation in Orbix

Orbix does not currently support persistent object storage for the reasons cited in Section 3.1.2.2.5. Hence, the implementation of the Data Store involves some custom development compatible with CORBA standards for distributed data objects. Shown in Figure 10 are the main elements to be considered in a Data Store (DS).

A key ingredient in the Data Store implementation is the Persistent Data Service (PDS). The PDS actually implements the mechanism for making data persistent and manipulating it. A specific PDS supports a protocol defining the way data is moved in and out of a particular type of object, and an interface to an underlying Data Store. The PDS is primarily responsible for translating from the object world to the storage world and vice versa.

Recall the discussion in Section 3.1.2.2.5 concerning the Orbix Object Fault Handler and the abstract class, *LoaderClass*. The instances of *LoaderClass*, which are called loaders, actually perform the services of the PDS. As illustrated in Figure 10, the loaders

are data-type-dependent and provide all the knowledge necessary to store and retrieve persistent data from/to non-volatile storage via the associated data servers. The loaders actually perform the translation of persistent data to objects and objects to persistent data.

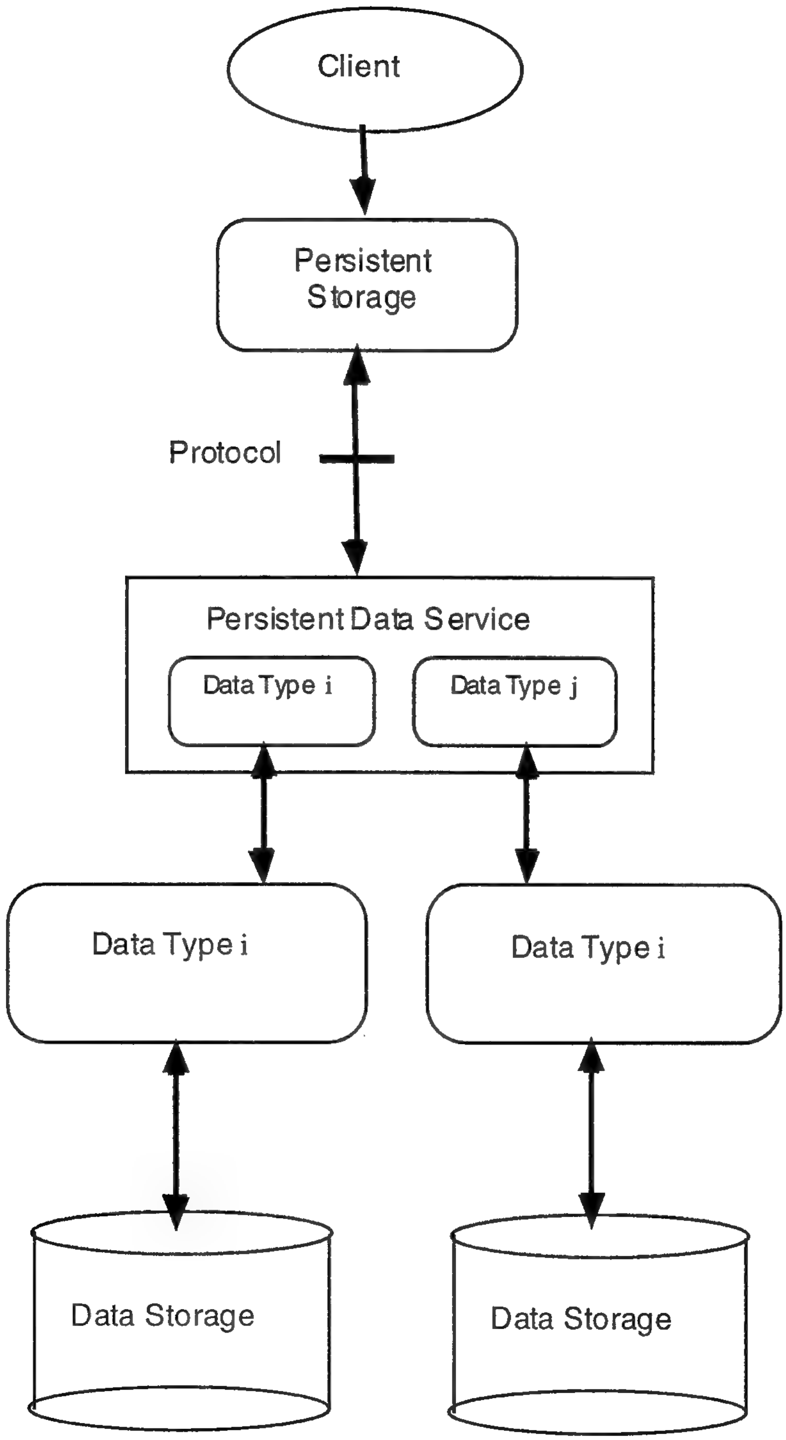

Figure 10. Main Elements in Data Store

3.1.2.4 CREST Data Library – Object Management Architecture

The objective of the final phase of the CREST program was to prototype a client-server-based heterogeneous data base management capability for the entire RSTER data located at the MHPCC facility. A key requirement was that data management services would be location independent (i.e., local vs. remote). An ORB-based architecture was prototyped as shown in Figure 11.

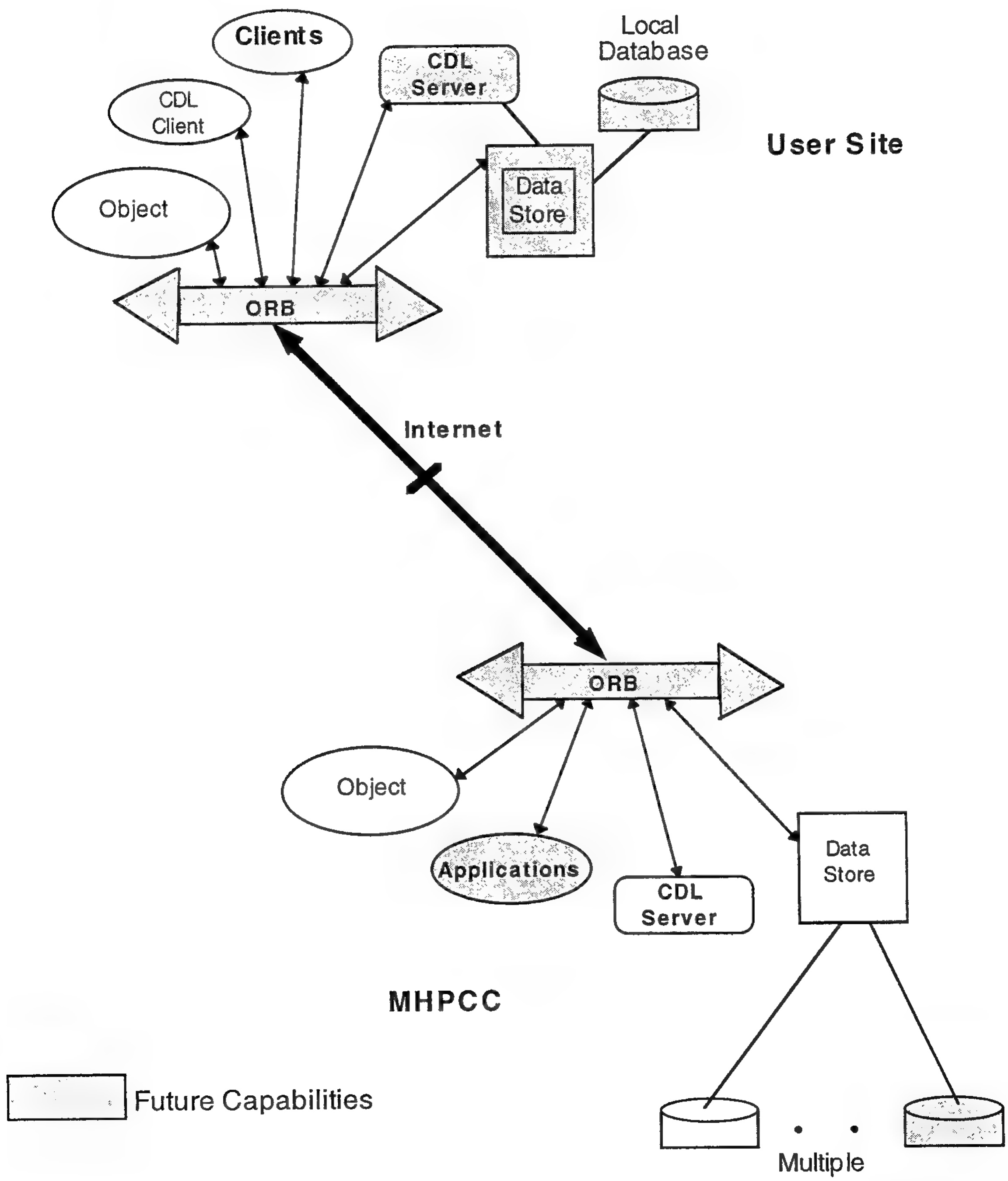

Figure 11. CREST Data Library ORB Architecture

The WWW client-server infrastructure is replaced with the OMG, OMA-compliant, IONA Orbix software architecture. The http protocol is replaced with the File Transfer Protocol, ftp, which is compatible with the Orbix communication layer. In the future, data management services will be provided at both the MHPCC site and the CDL user's local sites.

3.2 Rapid Prototyping

There are two main rapid prototyping methodologies: Throwaway and Evolutionary. In the throwaway model, the early prototype is literally discarded and not used in the final product. In the evolutionary model, all or parts of the earlier prototype stages are retained and used as building blocks during iteration towards the next level of development. The final version of the evolutionary prototype is delivered as the end product.

The CREST Data Library project is an example of both models. The initial prototype was developed using standard and reliable Web tools (i.e., *httpd and html*) to build the Web server and CDL query client. The CDL database server was constructed from a refined version of an advanced prototype imagery server that was previously developed for the Rome Laboratory IE2000 Facility. The server database metadata was revamped to conform to the requirements of the RSTER radar data. Utilizing these building block components, the initial prototype was rapidly constructed and fielded for operational use by users of the WWW.

The initial prototype was useful for proving the feasibility of the Crest Data Library concept. Although the CDL server proved to be adequate for accessing a single radar database, the scalability of this initial server to accommodate a wide variety of sensor databases at MHPCC, each with its particular metadata requirements, could not be demonstrated. Therefore, although the initial prototype can be considered throwaway, it will continue to be useful by serving as an introduction to the OMA-based Data Library. Another outcome of developing the initial prototype was the realization of the need for accessing heterogeneous databases. This realization gave greater credibility to the need for the ORB-based architecture that was to be developed in the final prototype.

3.3 Supporting and Future Technologies

One additional technology was incorporated into the CREST Data Library — tcl/tk (and the extension tix). Tcl/tk/tix is discussed in Section 3.3.1. In addition, there appears to be a role for Java, a new object oriented distributed programming language, in future improvements on the CREST Data Library. Section 3.3.2 provides an overview of Java.

3.3.1 tcl/tk/tix

Tcl stands for Tool Command Language and Tk is an extension to Tcl, and stands for Toolkit. Tk is an interface to the X Windows system, and Tix is a Motif-based widget set

that sits on top of Tk. Tcl 7.4, its associated X windows toolkit, Tk 4.0, and TIX 3.6 are currently (and freely) available on the Internet and may be directly used for application development. Tcl is both a scripting language and an interpreter for that language, which is designed for easily embedding into applications.

Tcl/Tk/Tix was heavily utilized in the application development for the final phase of the CREST Data Library project. The graphical user interface for the CDL query client application is the most graphical example of the Tcl/Tk/Tix implementation during this effort.

3.3.1.1 TCL

Tcl is both an interpreted language and a C library that may be embedded in application programs. The Tcl language is intended for issuing commands to interactive programs including shells. Tcl has a simple syntax and is programmable so that command procedures more powerful that the built-in command set can be written.

The Tcl scripting language is similar to other UNIX shell languages (e.g., C Shell, Bourne Shell, Perl, etc.), in that it permits complex scripts that execute other programs to configure new tools and extend existing applications. Tcl provides a standard syntax that is easy to learn and program. However, the ability to add a Tcl interpreter into an application is uniquely different than other shell languages. Thus, by adding a Tcl interpreter, applications may be structured as a set of primitive operations that are integrated by means of an executable script. Tcl scripts are ideal for gluing together separate pieces of C code and providing hooks for extendibility. The Tcl C library implements the Tcl interpreter and the scripting commands that implement variables, file I/O, control flow, and procedures. In addition, an application can define new Tcl commands, which are associated C or C++ procedures that the application provides. Accordingly, the Tcl script can call directly into the application by means of executing the application-defined Tcl commands. Conversely, an application can call Tcl scripts and set and query Tcl variables. The Tcl interpreter has been ported into UNIX, DOS, and Macintosh environments.

The Tcl library consists of a parser for the Tcl language, routines to implement the built-in Tcl commands, and several utility procedures that allow each application to extend Tcl with new commands specific to an application. The application provides application-specific commands plus procedures to collect commands for execution. The application program generates Tcl commands and passes them to the Tcl parser for execution. The Tcl commands may be collected either be inputting character strings from an input source, or by associating command strings with elements of the application's user interface (i.e., menu entries, buttons, etc.). When the Tcl library receives commands, the commands are parsed into component fields and built-in commands are executed directly. For commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases commands will invoke recursive invocations of the Tcl interpreter by

passing additional strings to execute (i.e., procedures, looping commands, and conditional commands). Figure 12 illustrates the Tcl command execution process and the interaction between the application program and the Tcl library.

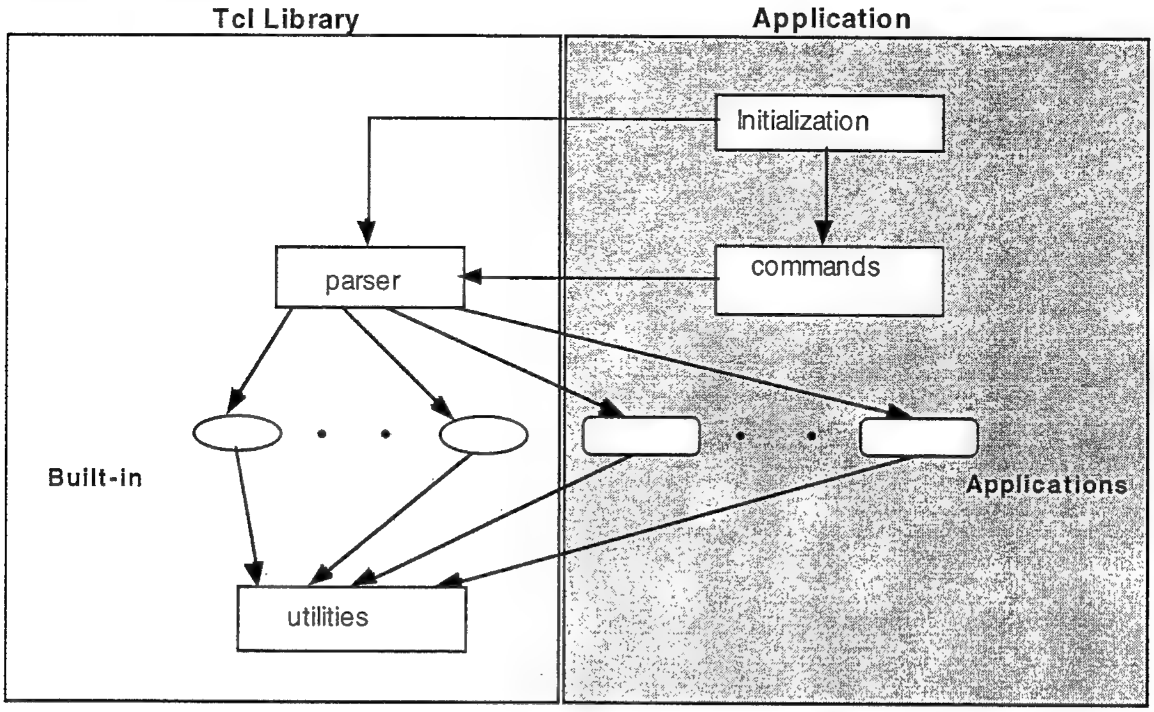

Figure 12. Tcl Command Interpretation and Execution

3.3.1.2 TK/TIX

The Tk toolkit is a Tcl extension (a group of new Tcl commands), which provides a Tcl interface to X Windows. Tk has been successfully compiled under X11 R4, R5, and R6. Tk greatly simplifies building a graphical user interface to an application. Since Tcl is an interpreted language, Tk-based interfaces tend to be much more customizable and dynamic than those built with a C- or C++-based X Windows toolkit. Tk implements the Motif "look and feel". The Tk toolkit defines Tcl commands that permit a programmer to create and manipulate user interface widgets. A robust set of Tk widgets are generally available via the TIX widget set, and the programmer may create custom Tk widgets in the C programming language.

Tk permits rapid development of graphical user interfaces for Tcl-based applications. Tk also provides a mechanism by which one application can send Tcl scripts to other Tk-based applications running on the same display, thus facilitating easy cooperation (interoperability) between tools. The common Tcl language framework makes it easier for applications to communicate with one another. As such, Tk is a productivity tool for user interface development.

3.3.2 Java

Java is a general-purpose object-oriented programming language for distributed object applications, developed by Sun. For future expansion and development of CREST Data Library's capabilities, Java and C++ must be closely examined and compared before a final commitment to either programming language is made. The experience gained from previous use of the ORBIX software in developing the ORB architecture prototype will be applied to an analysis of Java's capabilities in this area. In particular, the ease with which clients and object classes were developed using C++ will be compared to Java's use in their development.

Java Programming Language

The Java language provides an architecture-neutral, distributed, portable, interpreted, object-oriented programming language that is designed to be very similar to C++ and Objective C, but it omits many of the rarely used and confusing features of C and C++. Thus, Java is small, simple, familiar and convenient to use; and it provides automatic memory management, which greatly simplifies the object-oriented programming task. Java is also very small; the size of the basic interpreter, class support, standard libraries, and microkernel is less than 250 Kbytes. The basic library provides routines for applications that deal with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across a network via URLs with a degree of ease comparable to accessing a local file system.

Java is designed to support applications that are architecture-neutral and portable across distributed heterogeneous networks comprised of a wide variety of operating systems and other programming languages. To accommodate this diversity, the Java compiler generates bytecodes, which is an architecture-neutral intermediate format, designed to transport code efficiently to heterogeneous network nodes. The Java language bytecodes may be loaded, verified, and interpreted on any platform capable of running the Java run-time environment. With Java, the same version of the application runs on all platforms. The bytecodes can also be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. Figure 13 illustrates the flow of data and control from the Java language source code through the Java compiler, to the bytecode loader, bytecode verifier and then on to the Java interpreter. Note, the bytecode stream may be generated locally or from some other external system. The bytecode verifier ensures that the code passed to the interpreter will execute securely. The interpreter does not have to check anything and is able to run fast and efficiently.

The run-time performance of bytecodes converted to machine code is nearly as fast as native C or C++. Accordingly, Java code is appropriate for interactive and embedded real-time systems with diverse platform environments (i.e., UNIX, Windows, Windows NT, or Macintosh).

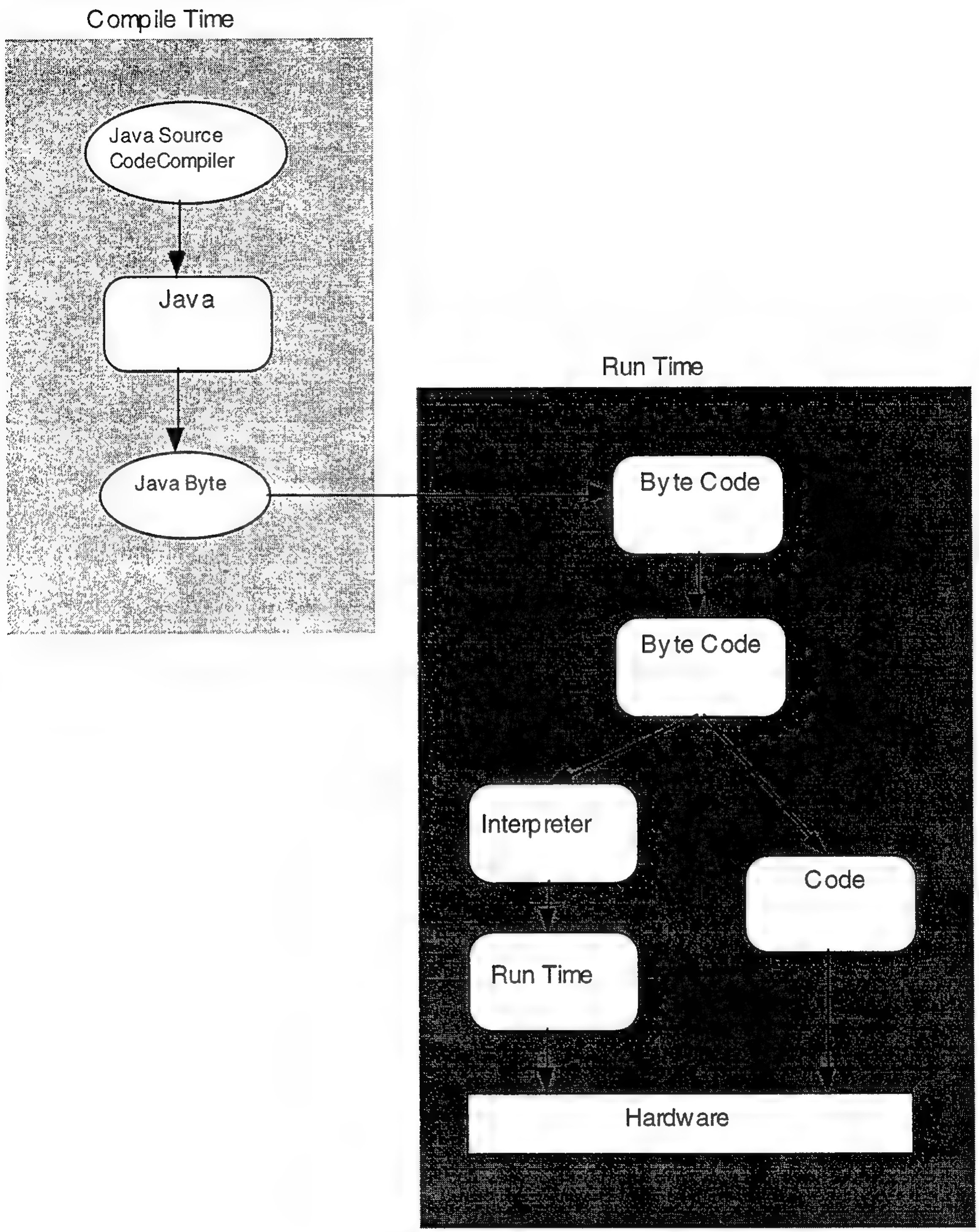

Figure 13. Java Source To Byte Code Data and Control Flow

Java facilitates writing of reliable programs by using a pointer methodology that prevents overwriting memory and corrupting data. Java programs do not use pointers explicitly. Objects are accessed by means of handles, which are similar to pointers, but Java does not support pointer arithmetic on object handles. However, Java supports true

arrays and subscript checking; it also uses extensive dynamic checking during interpretation. Because Java is intended to be used in networked/distributed environments, it was designed with security in mind. Java's authentication techniques are based on public-key encryption to enable the construction of tamper-free systems. In addition, since Java is a strongly-typed language, any changes to the semantics of pointers prevent applications from forging access to data structures or private data in objects that they have access to; hence, inhibiting virus activities.

Java is a much more flexible and dynamic language than C++; it was designed to adapt to an evolving environment. Java was designed for late binding, which allows interconnections between modules to be established at runtime. Thus, class libraries (comprised of plug and play components) can freely add new methods and instance variables without affecting any of their clients. Also, unlike C++, Java can determine what type of object is being pointed to by querying for runtime type information; this cannot be done in C++.

Currently, the Java Developers Kit (JDK) Version 1.0 is in Beta 2 release stage. The JDK includes the following software components:

- Java compiler
- Java language runtime
- Java debugger API and command line debugger
- Java Applet Viewer - for running and testing applets (object instances).

The JDK V1.0 Beta 2 can run on Solaris 2.3 (or higher), Windows NT/95, and Macintosh OS. The JDK software can be downloaded from Internet, and is free of charge. Also, developers do not need permission (from Sun) to distribute applets or client applications written in Java.

The complete Java system includes several libraries of utility classes and methods of use for developers to create multi-platform applications. These libraries include the following:

- *java.lang* collection of language types that are always imported into any compilation unit. Declarations of *Object* (root of the class hierarchy), *Class*, fundamental classes, threads, exceptions, and wrappers for primitive data types may be found in this collection.
- *java.io* analogous to UNIX standard I/O library for streams and random-access files.
- *java.net* provides support for sockets, telnet interfaces, and URLs.
- *java.util* provides container and utility classes (i.e., *Dictionary*, *Stack*, etc.).
- *java.awt* Abstract Windowing Toolkit, which provides an abstract layer enabling Java applications to be ported from one window system to another; contains classes for common widgets and user interface artifacts.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997/509-127/47084

DISTRIBUTION LIST

addresses	number of copies
ROME LABORATORY/OCSA ATTN: STANLEY E. BOREK 26 ELECTRONIC PKY ROME, NY 13441-4514	13
PAR GOVERNEMENT SYSTEMS CORP ATTN: JOHN REILLY 8383 SENECA TURNPIKE NEW HARTFORD, NY 13413-4991	2
ROME LABORATORY/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: RAYMOND TADROS GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1
ATTN: WALTER HARTMAN WRIGHT LABORATORY/AAM, BLDG. 620 2241 AVIONICS CIRCLE, RM N3-F10 WRIGHT-PATTERSON AFB OH 45433-7333	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

ATTN: R.L. DENISON 1
WRIGHT LABORATORY/MLPO, BLDG. 651
3005 P STREET, STE 6
WRIGHT-PATTERSON AFB OH 45433-7707

WRIGHT LABORATORY/FIVS/SURVIAC 1
2130 EIGHTH STREET, BLDG 45, STE 1
WRIGHT-PATTERSON AFB OH 45433-7542

ATTN: GILBERT G. KUPERMAN 1
AL/CFHI, BLDG. 248
2255 H STREET
WRIGHT-PATTERSON AFB OH 45433-7022

AUL/LSAD 1
600 CHENNAULT CIRCLE, BLDG. 1405
MAXWELL AFB AL 36112-6424

US ARMY STRATEGIC DEFENSE COMMAND 1
CSSD-IM-PA
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

NAVAL AIR WARFARE CENTER 1
6000 E. 21ST STREET
INDIANAPOLIS IN 46219-2189

COMMANDING OFFICER 1
NCCOSC RDT&E DIVISION
ATTN: TECHNICAL LIBRARY,CODE 0274
53560 HULL STREET
SAN DIEGO CA 92152-5001

COMMANDER, TECHNICAL LIBRARY 1
4747000/C0223
NAVAIRWARCENWPNDIV
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS 2
COMMAND (PMW 178-1)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

SPACE & NAVAL WARFARE SYSTEMS COMMAND, EXECUTIVE DIRECTOR (PD13A) ATTN: MR. CARL ANDRIANI 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
COMMANDER, SPACE & NAVAL WARFARE SYSTEMS COMMAND (CODE 32) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
CDR, US ARMY MISSILE COMMAND RSIC, BLDG. 4484 AMSMI-RD-CS-R, DOCS REDSTONE ARSENAL AL 35898-5241	2
ADVISORY GROUP ON ELECTRON DEVICES SUITE 500 1745 JEFFERSON DAVIS HIGHWAY ARLINGTON VA 22202	1
REPORT COLLECTION, CIC-14 MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1
AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUITE C211 ARNOLD AFB TN 37389-3211	1
AFIWC/MSO 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
NSA/CSS K1 FT MEADE MD 20755-6000	1
PHILLIPS LABORATORY PL/TL (LIBRARY) 5 WRIGHT STREET HANSCOM AFB MA 01731-3004	1

THE MITRE CORPORATION 1
ATTN: E. LADURE
D460
202 BURLINGTON RD
BEDFORD MA 01732

OUSD(P)/DTSA/DUTD 2
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

PAR GOVERNMENT SYSTEMS CORP 1
8383 SENECA TURNPIKE
NEW HARTFORD, NY 13413-4991

PAR GOVERNMENT SYSTEMS CORP 2
ATTN: R. PATRICK O'CONNOR
8383 SENECA TURNPIKE
NEW HARTFORD, NY 13413-4991

PAR GOVERNMENT SYSTEMS CORP 2
ATTN: AUDREY COPPERWHEAT
8383 SENECA TURNPIKE
NEW HARTFORD, NY 13413-4991

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.